中山大學 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

《Rust语言与内存安全设计》

第6讲 面向对象的Rust、错误处理

课程负责人：陈文清 助理教授

chenwq95@mail.sysu.edu.cn

2023年10月10日

面向对象的Rust基础

1. 类型系统
2. 泛型
3. 特征

特征（trait）例子

- **特征的继承，某个特征依赖其他特征**
- **从另一个案例来看：**
  - 我们要构建一个特斯拉TeslaRoadster对象，具有Vehicle和Car特征。

```rust
trait Vehicle {
    fn get_price(&self) -> u64;
}

trait Car: Vehicle {
    fn model(&self) -> String;
}
```

```rust
struct TeslaRoadster {
    model: String,
    release_date: u16
}

impl Car for TeslaRoadster {
    fn model(&self) -> String {
        "Tesla Roadster I".to_string()
    }
}
```

特征（trait）例子

- **特征的继承，某个特征依赖其他特征**
- **从另一个案例来看：**
  - ➢ 我们要构建一个特斯拉TeslaRoads

```rust
trait Vehicle {
    fn get_price(&self) -> u64;
}

trait Car: Vehicle {
    fn model(&self) -> String;
}
```

```rust
struct TeslaRoadster {
    model: String,
    release_date: u16
}

impl Car for TeslaRoadster {
    fn model(&self) -> String {
        "Tesla Roadster I".to_string()
    }
}
```

报错信息：

```
error[E0277]: the trait bound `TeslaRoadster: Vehicle` is not satisfied
  --> src/main.rs:22:6
   |
22 | impl Car for TeslaRoadster {
   |      ^^^ the trait `Vehicle` is not implemented for `TeslaRoadster`
   |
```

特征（trait）例子

- **特征的继承，某个特征依赖其他特征**
- **从另一个案例来看：**
  - 我们要构建一个特斯拉TeslaRoad

```
trait Vehicle {
    fn get_price(&self) -> u64;
}

trait Car: Vehicle {
    fn model(&self) -> String;
}
```

添加Vehicle特征实现：

```
struct TeslaRoadster {
    model: String,
    release_date: u16
}

impl Vehicle for TeslaRoadster {
    fn get_price(&self) -> u64 {
        200_000
    }
}

impl Car for TeslaRoadster {
    fn model(&self) -> String {
        "Tesla Roadster I".to_string()
    }
}
```

面向对象的Rust进阶

1. 类型系统
2. 泛型
3. 特征
4. 包含泛型的特征
5. 标准库特征
6. 生命周期

4. 包含泛型的特征——特征区间（trait bound)

- trait as parameters

```rust
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

4. 包含泛型的特征——特征区间（trait bound）

- trait as parameters
- trait bound

```rust
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

```rust
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

impl Trait 语法适用于简单的情况，但实际上是称为trait bound的语法糖

4. 包含泛型的特征——特征区间（trait bound)

- 以另一个案例来看，简单的泛型加法也需要trait bound

```
fn add_thing<T>(fst: T, snd: T) {
    let _ = fst + snd;
}

fn main() {
    add_thing(2, 2);
}
```

```
fn add_thing<T: std::ops::Add>(fst: T, snd: T) {
    let _ = fst + snd;
}

fn main() {
    add_thing(2, 2);
}
```

4. 包含泛型的特征——特征区间（trait bound)

- 指示trait bound的另一种用法，where语句，增强可读性

```rust
fn add_thing<T: std::ops::Add>(fst: T, snd: T) {
    let _ = fst + snd;
}

fn main() {
    add_thing(2, 2);
}
```

```rust
fn add_thing<T>(fst: T, snd: T)
where T: std::ops::Add
{
    let _ = fst + snd;
}
fn main() {
    add_thing(2, 2);
}
```

4. 包含泛型的特征——特征区间（trait bound）

■ 4.1 类型上的特征区间

```rust
use std::fmt::Display;
struct Foo<T: Display> {
    bar: T
}
// or
struct Bar<F> where F: Display {
    inner: F
}
fn main() {}
```

例如上述代码，给结构体的泛型添加trait bound；

（不鼓励，因为对类型自身施加了限制，

一般的做法是在函数或者方法中添加trait bound ）

4. 包含泛型的特征——特征区间（trait bound）

■ 4.2 使用“+”将特征组合为区间

```rust
// traits_composition.rs
trait Eat {
    fn eat(&self) {
        println!("eat");
    }
}
trait Code {
    fn code(&self) {
        println!("code");
    }
}
trait Sleep {
    fn sleep(&self) {
        println!("sleep");
    }
}
```

```rust
trait Programmer : Eat + Code + Sleep {
    fn animate(&self) {
        self.eat();
        self.code();
        self.sleep();
        println!("repeat!");
    }
}
```

注意：为某个对象实现Programmer特征需要对其继承的特征也分别实现（ Eat、Code 、Sleep）。

5. 标准库特征

■ 标准库自带了一些内置的特征

```rust
use std::ops::Add;

#[derive(Default, Debug, PartialEq, Copy, Clone)]
struct Complex<T> {
    // Real part
    re: T,
    // Complex part
    im: T
}
```

```rust
let second: Complex<i32> = Complex::default();
println!("{:?}", second);
```

自动派生内置特征；

接下来手动实现Add

5. 标准库特征

■ **标准库自带了一些内置的特征**

```
use std::ops::Add;

#[derive(Default, Debug, PartialEq, Copy, Clone)]
struct Complex<T> {
    // Real part
    re: T,
    // Complex part
    im: T
}
```

```
impl<T: Add<T, Output=T>> Add for Complex<T> {
    type Output = Complex<T>;
    fn add(self, rhs: Complex<T>) -> Self::Output {
        Complex { re: self.re + rhs.re, im: self.im + rhs.im }
    }
}
```

高亮部分表示复数中的实部和虚部需要符合Add特征，<T, Output=T>表示Add特征需具有相同类型的输入和输出

6. 生命周期

■ 生命周期

生命周期（lifetime）概念：编译器（中的借用检查器）用它来保证所有的引用都是有效的。

Every reference in Rust has a *lifetime*, which is the scope for which that reference is valid

https://rustwiki.org/zh-CN/rust-by-example/scope/lifetime.html

https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html

6. 生命周期

■ 生命周期

```rust
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

Q：能否编译通过？

6. 生命周期

■ 生命周期

```rust
fn main() {
    let r;                // ----------+-- 'a
                          //           |
    {                     //           |
        let x = 5;        // -+-- 'b   |
        r = &x;           //  |        |
    }                     // -+        |
                          //           |
    println!("r: {}", r); //           |
}                         // ----------+
```

Rust编译器中的borrow checker可以检查变量的生命周期

在这里，我们用 'a 注释了 r 的生命周期，用 'b 注释了 x 的生命周期。

当print时，r指向的x已经超出生命周期，报错！

6. 生命周期

■ 生命周期

```rust
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```rust
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
```

Q：能否编译通过？

6. 生命周期

■ 生命周期

```rust
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```rust
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
```

A： 不能，上述案例无法确定返回值的生命周期与**x**还是与**y**一致

6. 生命周期

■ 生命周期

生命周期（lifetime）概念：编译器（中的借用检查器）用它来保证所有的借用都是有效的。

Every reference in Rust has a *lifetime*, which is the scope for which that reference is valid

- 添加生命周期标识符（一种特殊的泛型）： ‘a
- 确保运行时使用的实际引用肯定有效
- 除了‘a，还可以使用其他字母（‘b），也可以使用更长的描述性名称（‘ctx，‘reader等）
- 关键词**static**修饰的生命周期，在程序运行期间都有效：

```
fn main() {
    let _a: &'static str = "I live forever";
}
```

6. 生命周期

■ 生命周期

Q：什么情况下需要添加生命周期标识符（ **Lifetime Annotation Syntax**）？

A：通过添加生命周期标识符，显式指定引用变量的生命周期：

```rust
fn longest<'a>(x: &'a str, y: &'a str)
-> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```rust
fn longest<'a, 'b>(x: &'a str, y: &'b
str) -> &'a str {
    x
}
```

```rust
    //codes in main()
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
```

6. 生命周期

■ 结构体中的生命周期

```rust
struct SomeRef<T> {
    part: &T
}

fn main() {
    let a = SomeRef { part: &43 };
}
```

报错信息：

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:2:11
  |
2 |     part: &T
  |           ^ expected named lifetime parameter
  |
```

包含引用的结构，需要生命周期注释

6. 生命周期

■ 结构体中的生命周期

生命周期（lifetime）概念：编译器（中的借用检查器）用它来保证所有的借用都是有效的。

Every reference in Rust has a *lifetime*, which is the scope for which that reference is valid

```rust
struct SomeRef<T> {
    part: &T
}

fn main() {
    let a = SomeRef { part: &43 };
}
```

→

```rust
struct SomeRef<'a, T> {
    part: &'a T
}

fn main() {
    let a = SomeRef { part: &43 };
}
```

- 添加生命周期标识符（一种特殊的泛型）： ‘a
- to ensure the actual references used at runtime will definitely be valid
- 除了‘a，还可以使用其他字母（‘b），也可以使用更长的描述性名称（‘ctx, ‘reader等）

6. 生命周期

■ 什么时候可以省略生命周期注释?

```rust
fn first_word<'a>(s: &'a str) -> &'a str {
```

```rust
fn first_word(s: &str) -> &str {
    s
}
```

编译器能够推断出返回值的生命周期：可以省略

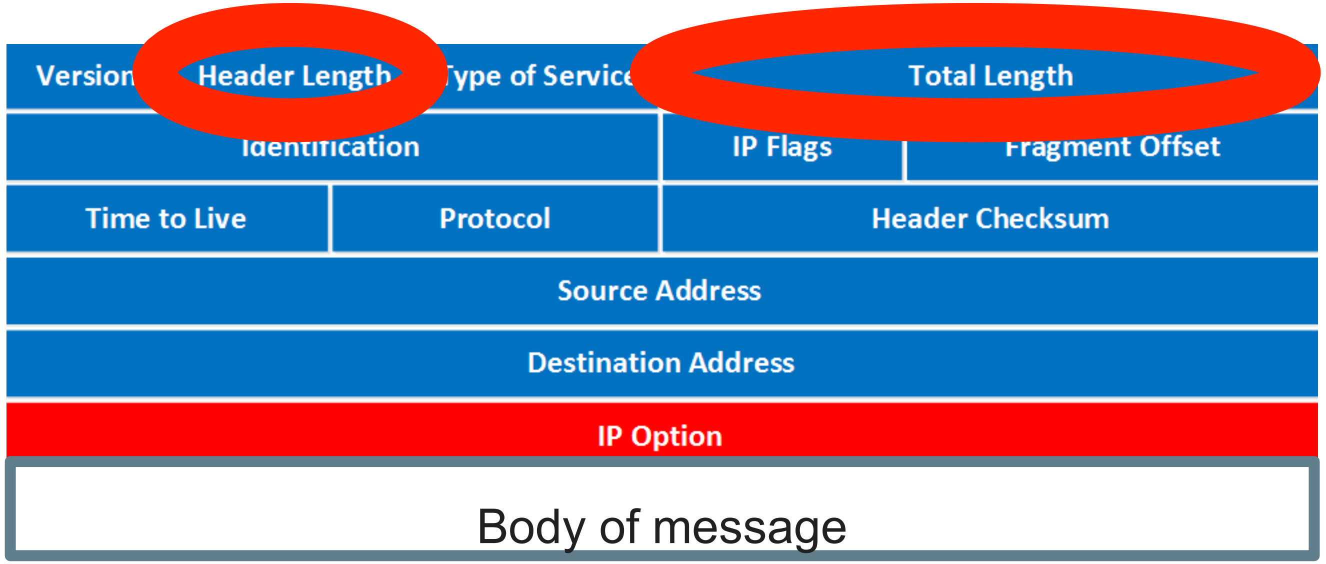
小结

- 类型是静态语言最棒的特性，允许用户在编译期间表达丰富的内容；
- 类型、泛型和特征，对于如何复用代码而言是最重要的；

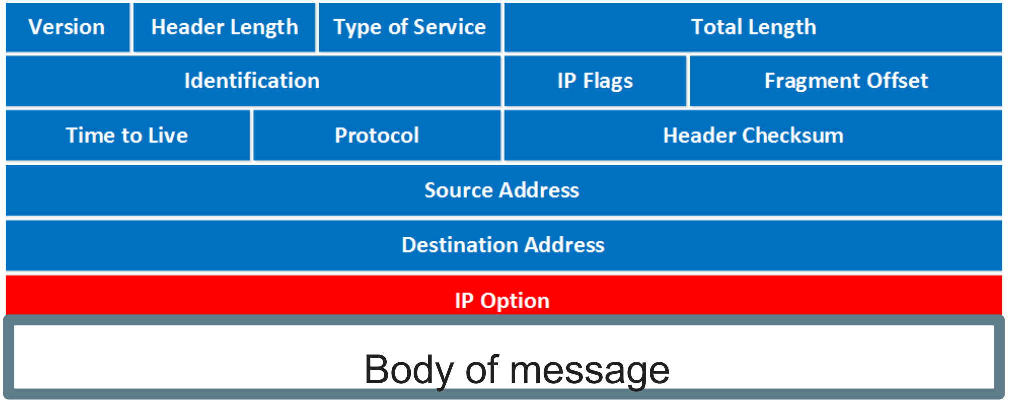
错误处理

远程代码执行

- 想象一下服务器从网络接收消息
  - 与通过 Internet 传输的所有消息一样，它封装在 IP (IPv4) 标头中
  - IP 标头*可以*是可变长度的。 IP 标头的长度[应该]在“标头长度”字段中指定。
  - 整个消息的长度[应该]在“总长度”字段中指定。
- *请注意，任何人（例如攻击者！）都可以填充这些字段*



远程代码执行




```
struct message {
    ipv4_hdr iphdr;
    ipv4_options[MAX_IP_OPTIONS] opts;
    char[MAX_DATA_LEN] data;
}
```

```
/* Given: read-only copy of entire message, read in from
   the network. */
void* process_and_return_data(const struct message *msg) {

    // Allocate space for local, mutable copy.
    void *local_copy = malloc(get_len(msg));

    // Copy only the body of the message
    memcpy(local_copy + get_hdr_len(msg->iphdr),
           msg + get_hdr_len(msg->iphdr),
           length_of_body);

    process_data(local_copy + get_hdr_len(msg->iphdr));

    // Copy in IP hdr
    memcpy(local_copy, msg, get_hdr_len(msg->iphdr));

    return local_copy;
}
```

远程代码执行

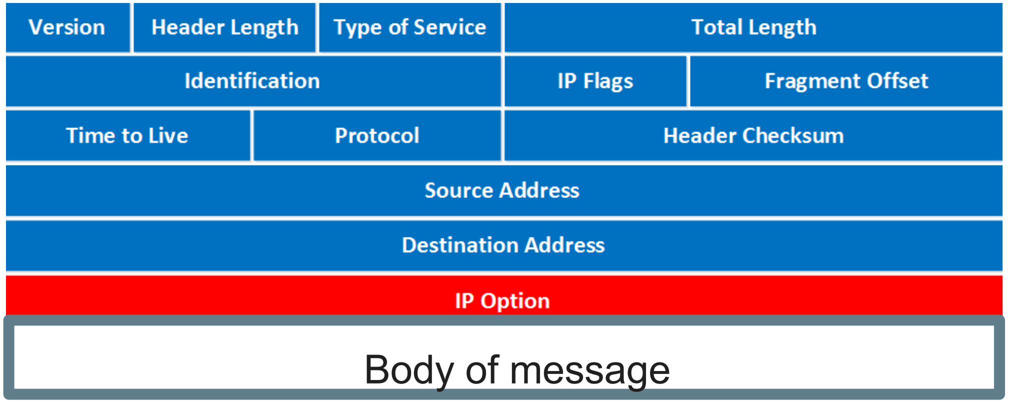
The **malloc**() function allocates *size* bytes and returns a pointer to the allocated memory. *The memory is not initialized*. If *size* is 0, then **malloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

type. On error, these functions return NULL. NULL may also be returned by a successful call to **malloc**() with a *size* of zero, or

calloc(), **malloc**(), **realloc**(), and **reallocarray**() can fail with the following error:

ENOMEM Out of memory. Possibly, the application hit the **RLIMIT_AS** or **RLIMIT_DATA** limit described in getrlimit(2).

远程代码执行




```
struct message {
    ipv4_hdr iphdr;
    ipv4_options[MAX_IP_OPTIONS] opts;
    char[MAX_DATA_LEN] data;
}
```

```
/* Given: read-only copy of entire message, read in from
   the network. */
void* process_and_return_data(const struct message *msg) {

    // Allocate space for local, mutable copy.
    void *local_copy = malloc(get_len(msg));

    // Copy only the body of the message
    memcpy(local_copy + get_hdr_len(msg->iphdr),
           msg + get_hdr_len(msg->iphdr),
           length_of_body);

    process_data(local_copy + get_hdr_len(msg->iphdr));

    // Copy in IP hdr
    memcpy(local_copy, msg, get_hdr_len(msg->iphdr));

    return local_copy;
}
```

Key insight: `malloc` could fail and return NULL

`local_copy + [value]` could be… anything.

问题

- 缺乏适当的错误处理
- 使用 NULL 代替实际值

Important note but not really related to what we're talking about today: you should never ever EVER trust values that come from the network!

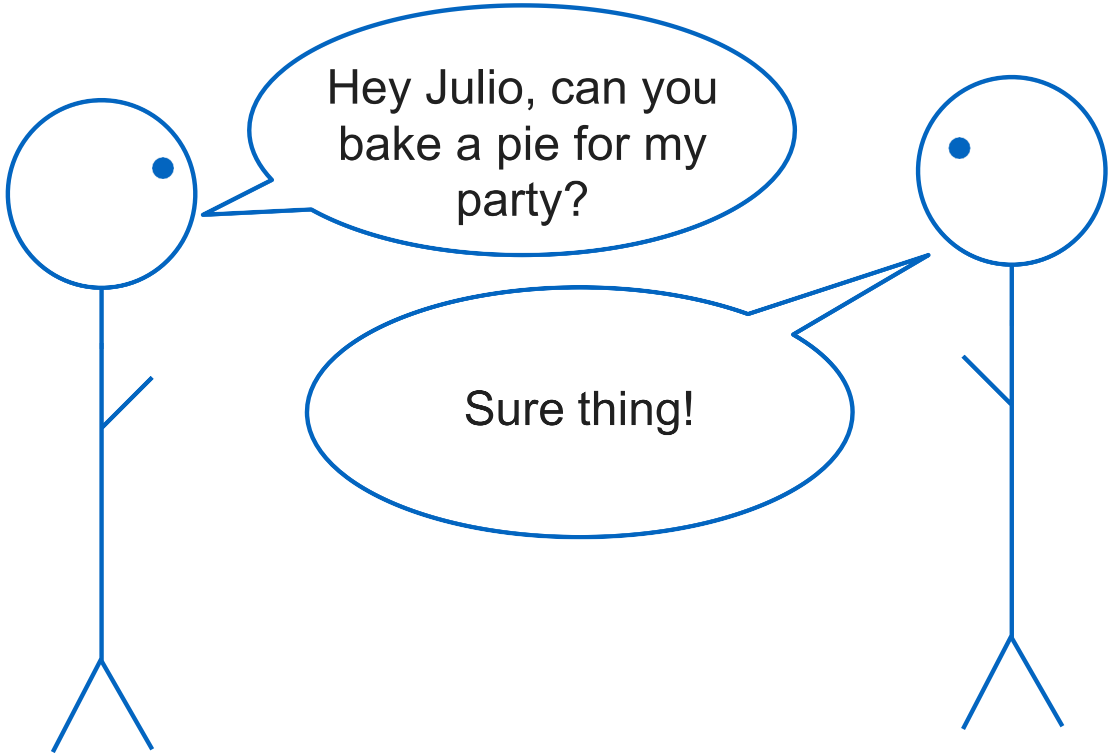
```c
/* Given: read-only copy of entire message, read in from
   the network. */
void* process_and_return_data(const struct message *msg) {

    // Allocate space for local, mutable copy.
    void *local_copy = malloc(get_len(msg));

    // Copy only the body of the message
    memcpy(local_copy + get_hdr_len(msg->iphdr),
           msg + get_hdr_len(msg->iphdr),
           length_of_body);

    process_data(local_copy + get_hdr_len(msg->iphdr));

    // Copy in IP hdr
    memcpy(local_copy, msg, get_hdr_len(msg->iphdr));

    return local_copy;
}
```

Handling errors

C 中的错误处理

- 如果函数可能遇到错误，则其返回类型将设为 int （或有时为 void*）。
- 如果函数成功，则返回 0。否则，如果遇到错误，则返回 -1。 （如果函数返回指针，则在成功情况下返回有效指针，如果发生错误则返回 NULL。）
- 遇到错误的函数将全局变量 errno 设置为一个整数，指示出了什么问题。 如果调用者发现函数返回-1或NULL，它可以检查errno以查看遇到了什么错误。

C 中的错误处理



int main()

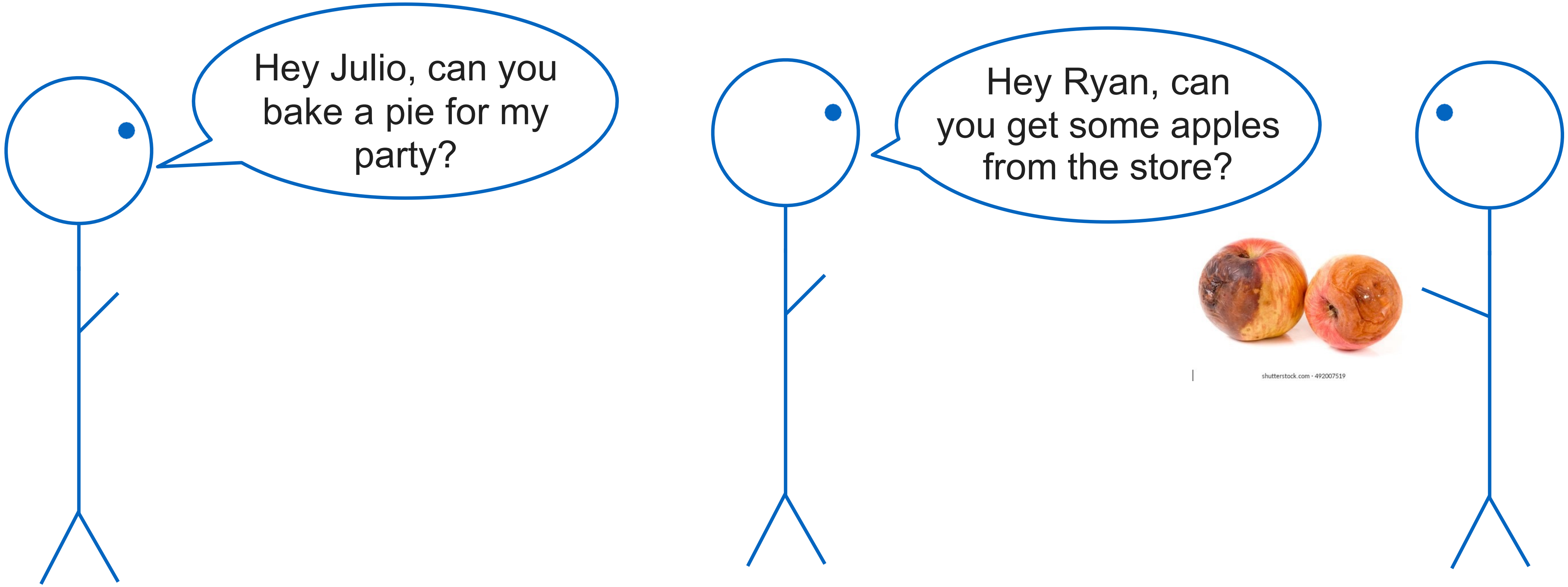
julio:

```
struct apple_pie *make_pie()
    { get_apples();
    bake_ingredients();
}
```

ryan:

```
struct apple *get_apples()
```

C 中的错误处理



int main()

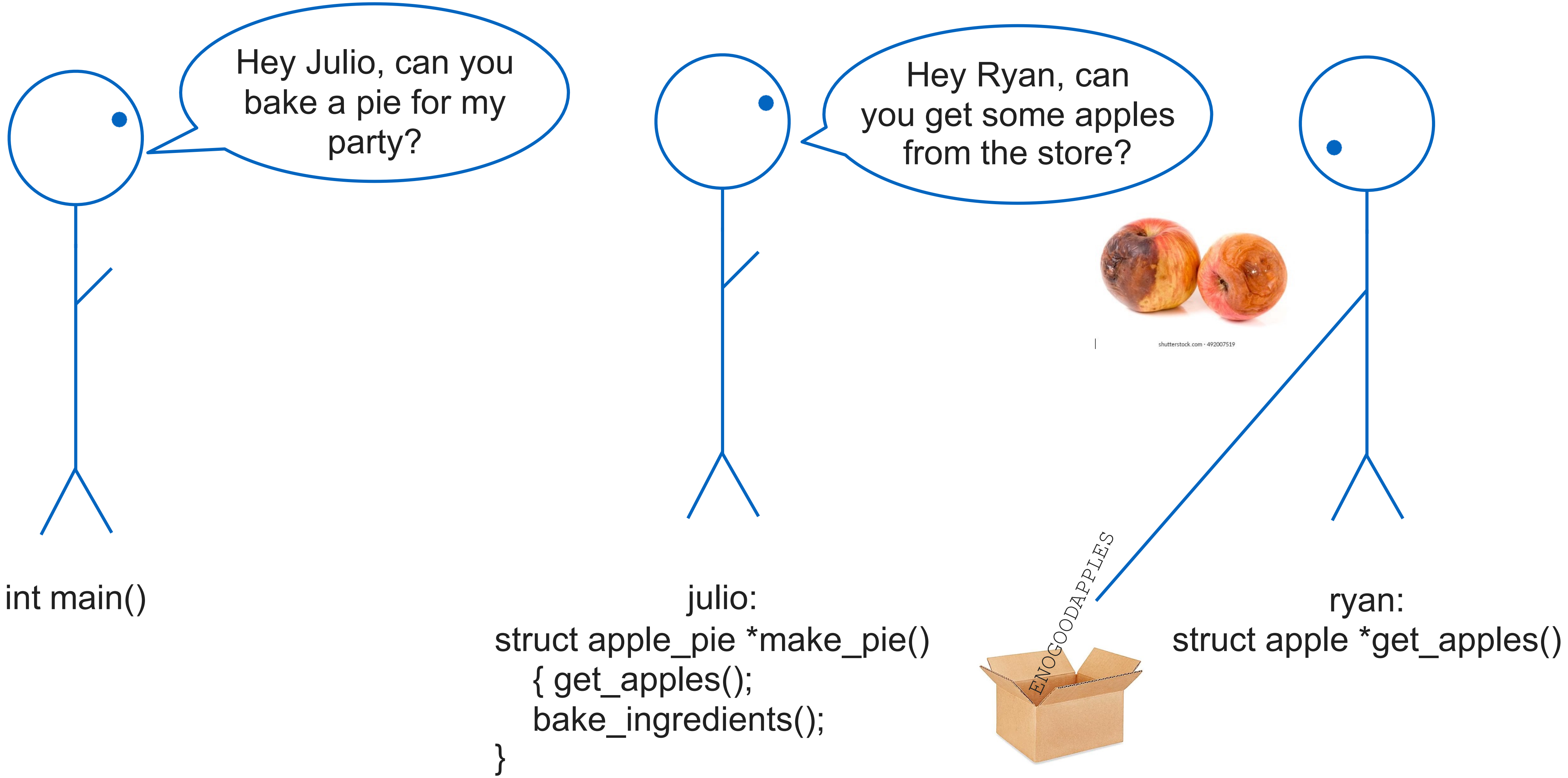
julio:

```
struct apple_pie *make_pie()
    { get_apples();
    bake_ingredients();
}
```

ryan:

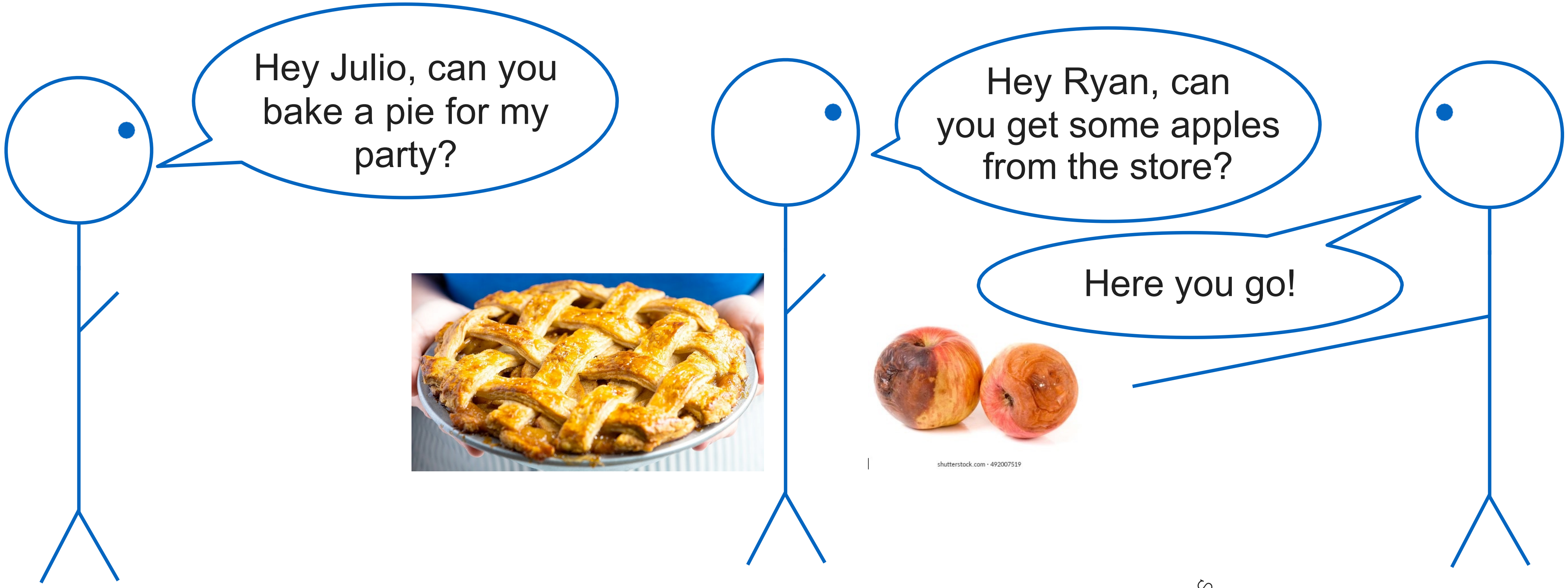
```
struct apple *get_apples()
```

C 中的错误处理



Hey Julio, can you bake a pie for my party?
Hey Ryan, can you get some apples from the store?
ENOGOODAPPLES
int main()
julio:
struct apple_pie *make_pie()
{ get_apples();
bake_ingredients();
}
ryan:
struct apple *get_apples()

C 中的错误处理



int main()

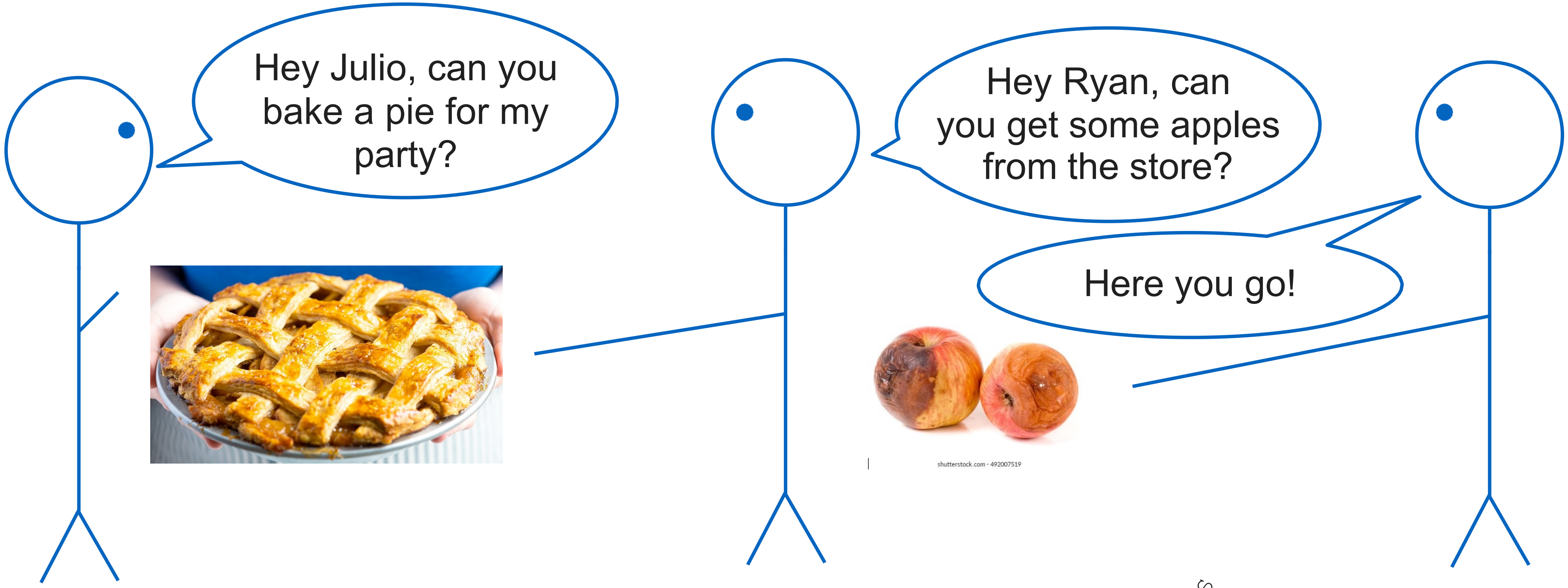
julio:

```
struct apple_pie *make_pie()
  { get_apples();
    bake_ingredients();
}
```

ryan:

```
struct apple *get_apples()
```

C 中的错误处理



int main()

julio:

```
struct apple_pie *make_pie()
    { get_apples();
    bake_ingredients();
}
```

ryan:

```
struct apple *get_apples()
```

Broken code from earlier

```
/* Given: read-only copy of entire message, read in from
   the network. */
void* process_and_return_data(const struct message *msg) {

    // Allocate space for local, mutable copy.
    void *local_copy = malloc(get_len(msg));

    // Copy only the body of the message
    memcpy(local_copy + get_hdr_len(msg->iphdr),
           msg + get_hdr_len(msg->iphdr),
           length_of_body);

    process_data(local_copy + get_hdr_len(msg->iphdr));

    // Copy in IP hdr
    memcpy(local_copy, msg, get_hdr_len(msg->iphdr));

    return local_copy;
}
```

Missing error check! 💣💣

CVE-2015-8812

- C关键的Linux内核漏洞：通过发送格式不正确的网络数据包，远程攻击者可以在内核中执行任意代码。
- 一组内核网络功能返回错误时为-1，成功时为0，但在“警告”时也返回其他值。
  - 如：当检测到拥塞时，返回NET_XMIT_CN（定义为2）。
- 调用这些功能的代码看到非零的返回代码，并假设存在网络错误。
- 释放了仍在用于网络的内存，导致使用后释放 + 双重释放（Use-after-free + double free）！

The fix

```diff
--- a/drivers/infiniband/hw/cxgb3/iwch_cm.c
+++ b/drivers/infiniband/hw/cxgb3/iwch_cm.c
@@ -149,7 +149,7 @@ static int iwch_l2t_send(struct t3cdev *tdev, struct sk_buff *skb, struct l2t_en
 	error = l2t_send(tdev, skb, l2e);
 	if (error < 0)
 		kfree_skb(skb);
-	return error;
+	return error < 0 ? error : 0;
 }
```

😰

Key insight

- 不同的返回值可能性表明成功+不同类型的错误（这确实很常见）
- 记录在（例如）文档页面和/或标题注释中
- 所有这些都只是整数
- 调用者必须记住处理所有情况

Proper C error checking is ugly

- 程序员必须记住他们调用的函数是否可能返回错误

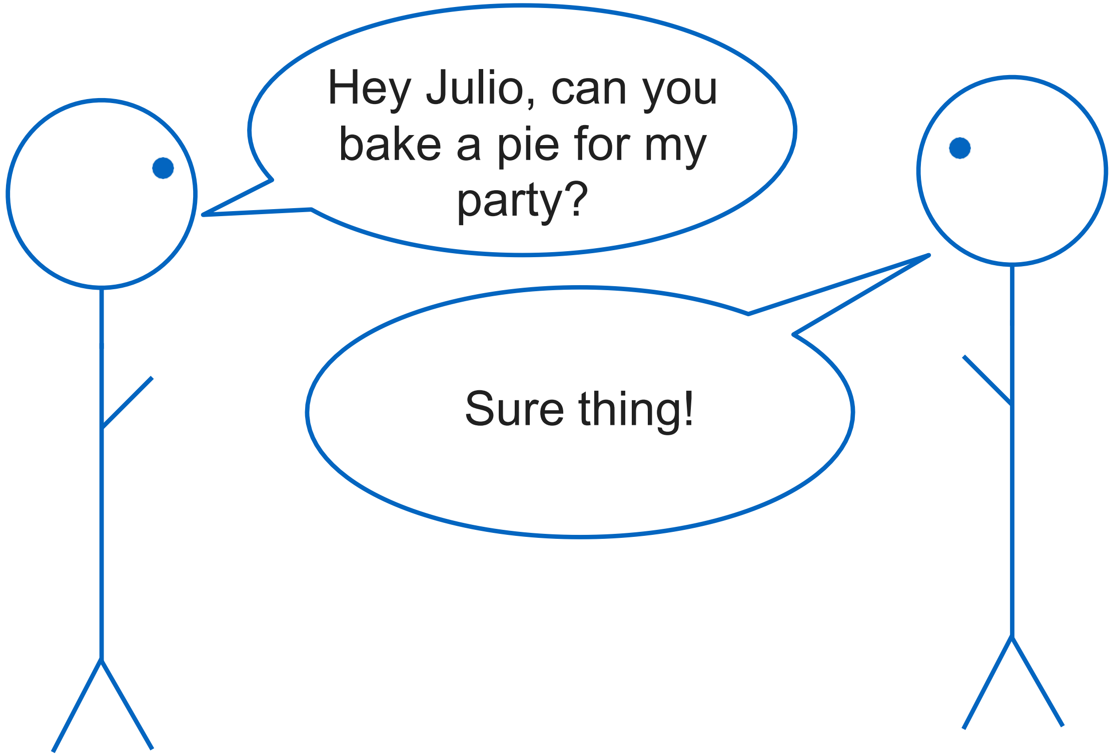
- 每次可能返回错误的函数调用后，必须检查是否发生错误并正确处理
    - This isn’t good enough:

```c
void *buf = malloc();
if (buf == NULL) {
   perror("error allocating memory");
}
memcpy(buf + offset, src, size);
```

- 使用 errno 处理特定错误可能会产生容易出错的 if 语句混乱
- [有时函数文档甚至没有正确记录可能返回的错误]()

C++（以及许多其他语言）中的错误处理

C++ 中的错误处理：异常

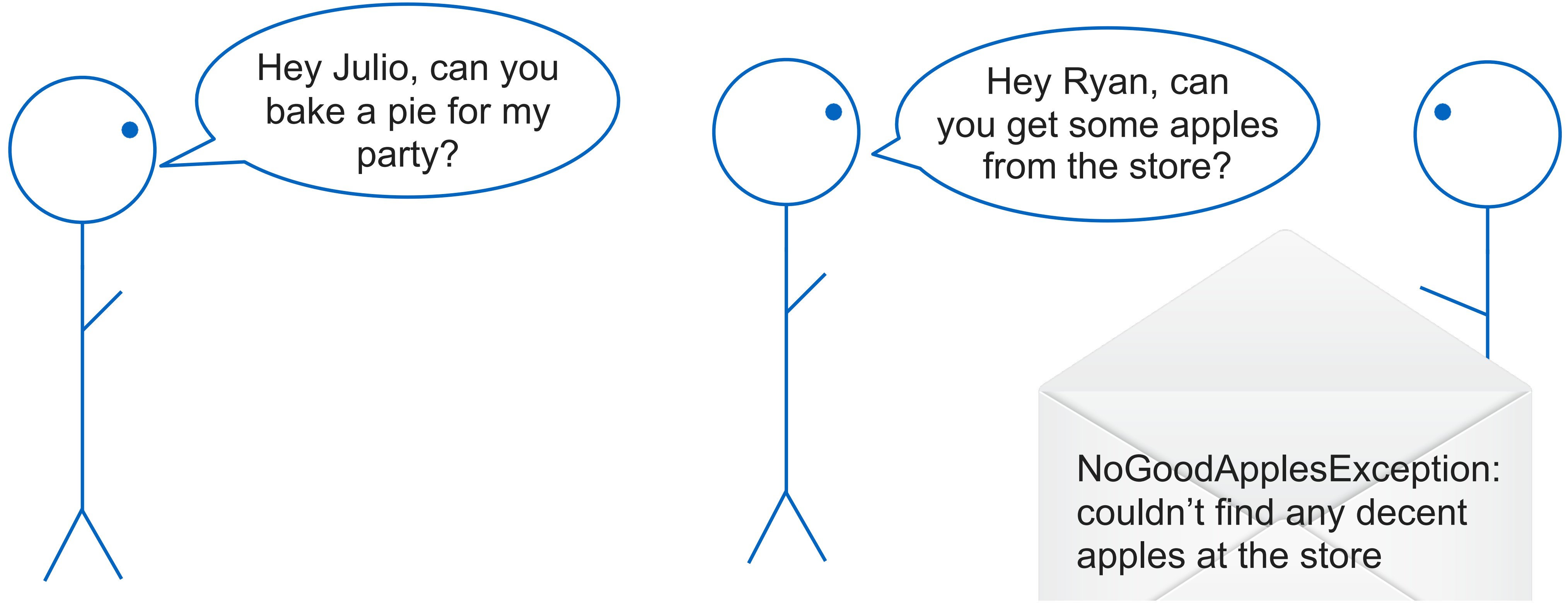


int main()

julio:
struct apple_pie *make_pie()
{ get_apples();
bake_ingredients();
}

ryan:
struct apple *get_apples()

C++ 中的错误处理：异常



int main()

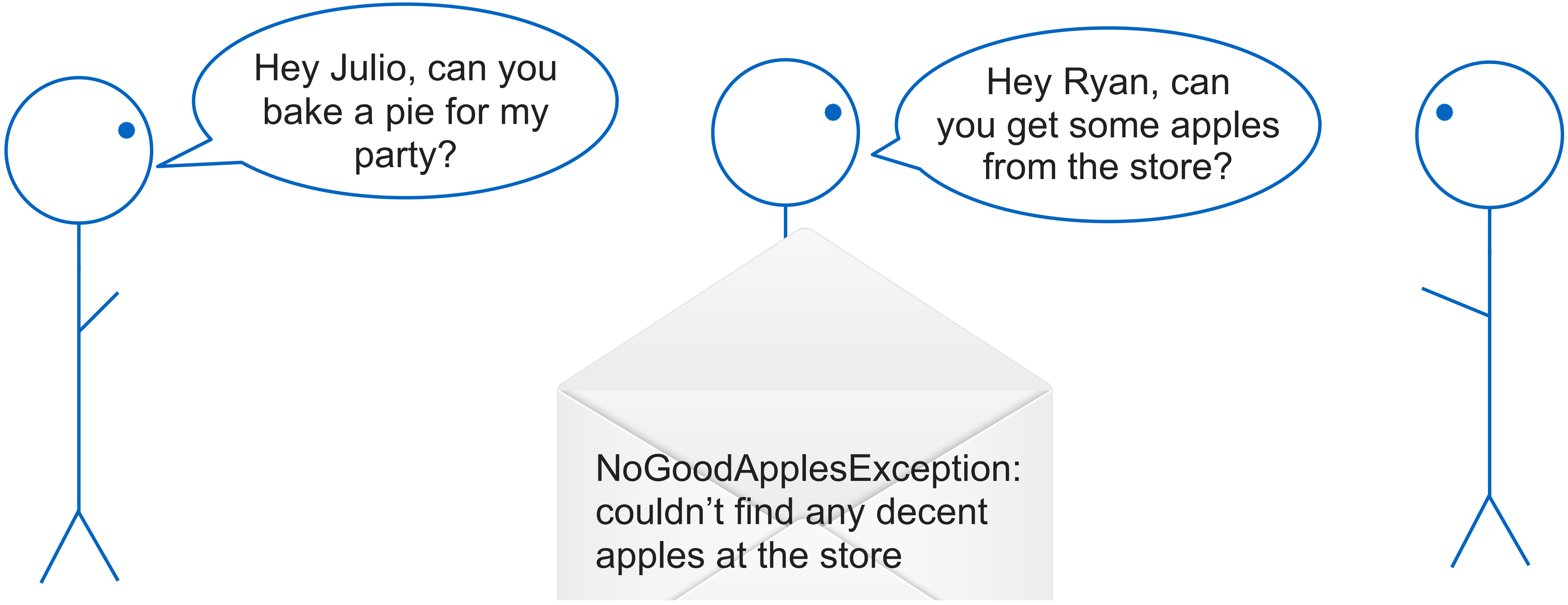
julio:

```
struct apple_pie *make_pie()
    { get_apples();
    bake_ingredients();
}
```

ryan:

```
struct apple *get_apples()
```

C++ 中的错误处理：异常

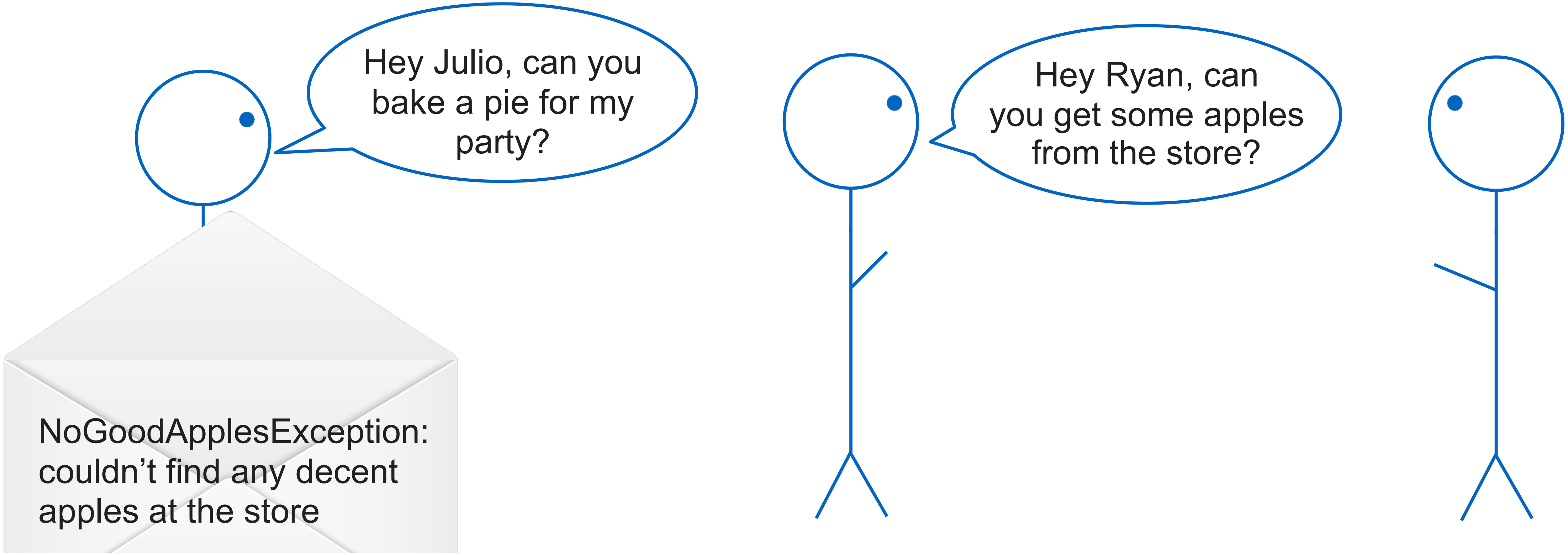


int main()

julio:
struct apple_pie *make_pie()
{ get_apples();
bake_ingredients();
}

ryan:
struct apple *get_apples()

C++ 中的错误处理：异常



int main()

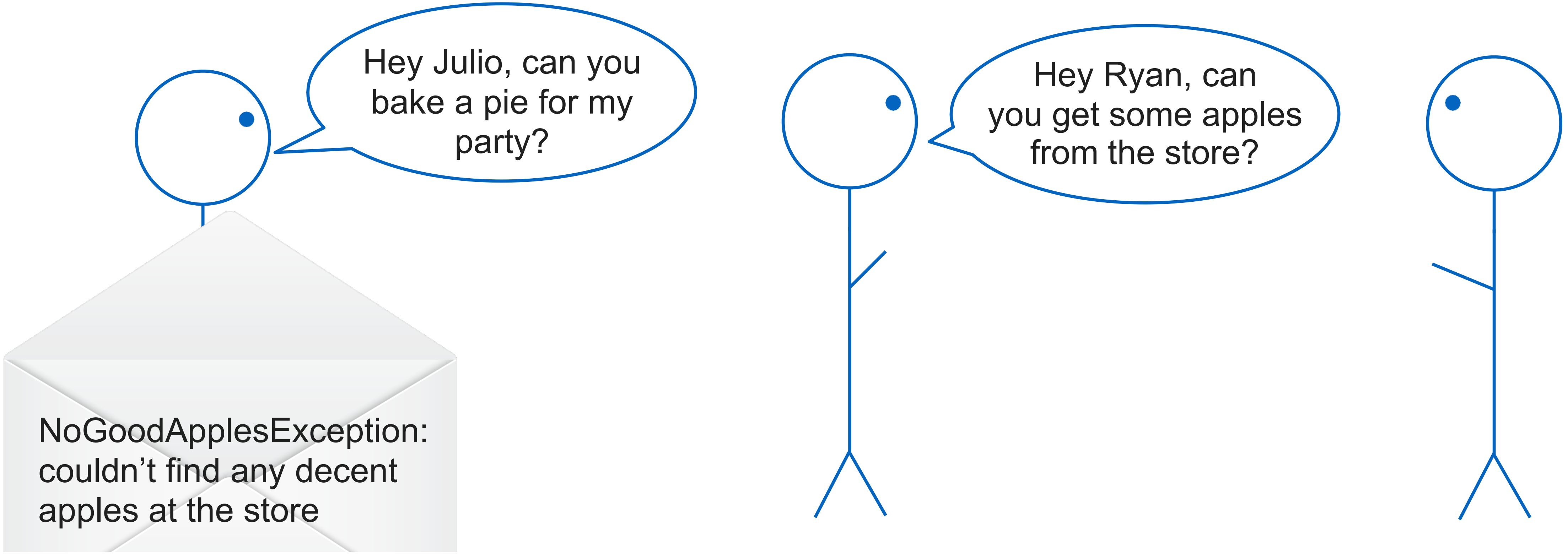
julio:

```
struct apple_pie *make_pie()
    { get_apples();
    bake_ingredients();
}
```

ryan:

```
struct apple *get_apples()
```

C++ 中的错误处理：异常




```
int main() {
  try {
    make_pie();
  } catch(NoGoodApplesException &e) {
    party_without_pie();
}
```

julio:

```
struct apple_pie *make_pie()
  { get_apples();
  bake_ingredients();
}
```

ryan:

```
struct apple *get_apples()
```

对 C 风格错误处理的巨大改进

- 您不必在每次调用可能产生错误的函数时都编写错误传播代码
  - 异常会自动在堆栈中传播，直到被 try/catch 处理为止
- 错误不会被忽视
  - 最坏的情况是，它们会传播到 main() 并导致程序崩溃
  - 听起来很糟糕，但是崩溃比程序继续在未定义的状态下运行要好得多

Except Exceptions

- 为什么异常（ Exceptions ）可能不那么热门?
  - ➢ 故障模式变得难以推理：任何函数都可以随时抛出任何异常；
  - ➢ 代码可能会因完全不相关的函数抛出异常而失败；
  - ➢ 随着新错误的添加，不断发展的代码库变得更加难以管理，很难发现哪里可能发生错误

- 可能导致资源泄漏和其他意外行为
  - ➢ 由于这个原因，许多代码库都禁止异常

Exceptions without RAII: sad times

RAII（Resource Acquisition Is Initialization）= “资源获取即初始化” 。

在具有RAII的编程语言中，资源与对象绑定;

当对象被销毁时，资源会被释放。（例如：C++析构函数。）

Exceptions without RAII: sad times

```
void process_input() {
    char *buf = malloc(128);

    // read input from user:
    fgets(buf, 128, stdin);
    // do more processing on input:
    some_helper(input);

    free(buf);
}
```

Looks good to me?

```
int main() {
    while (true) {
        try {
            process_input();
        } catch (BadInputError) {
            cerr << "That wasn't valid, try again" << endl;
        }
    }
}
```

```
void some_helper(string input) {
    if (input == "uh oh") {
        throw BadInputError("I don't like that");
    }
}
```

Exceptions without RAII: sad times

RAII = "resource acquisition is initialization". In a language with RAII, resources are tied to an object; when object is destroyed, resources are freed. (Ex: C++ destructor.)

```
void process                                                         ndl;
    char *be

    // read
    fgets(bu
    // do mo
    some_h l
    e
          f
    free(bu
}
```

Looks go

Video link: https://twitter.com/c0dehard/status/1327718161848872960

Exceptions without RAII: sad times

```
void process_input() {
    char *buf = malloc(128);

    // read input from user:
    fgets(buf, 128, stdin);
    // do more processing on input:
    some_helper(input);

    free(buf);
}
```

Looks good to me?

MEMORY LEAK!!!!

```
int main() {
    while (true) {
        try {
            process_input();
        } catch (BadInputError) {
            cerr << "That wasn't valid, try again" << endl;
        }
    }
}
```

```
void some_helper(string input) {
    if (input == "uh oh") {
        throw BadInputError("I don't like that");
    }
}
```

Q & A

Thanks!